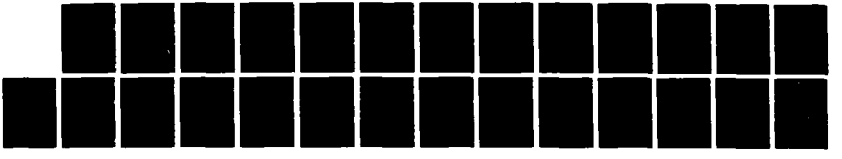
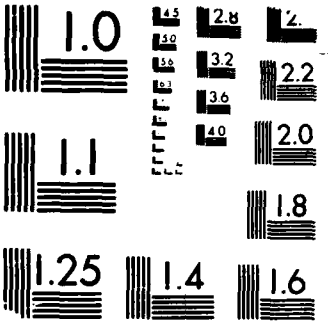
AD-A193 940 OPTIMISATION ALGORITHMS FOR HIGHLY PARALLEL COMPUTER ARCHITECTURES THE PE..(U) HATFIELD POLYTECHNIC (ENGLAND) L C DIXON ET AL. MAR 88 DAJA45-87-C-0038 1/1

UNCLASSIFIED F/G 12/6 NL





AD 1988

OPTIMISATION ALGORITHMS FOR HIGHLY PARALLEL COMPUTER ARCHITECTURES

SECOND INTERIM REPORT

THE PERFORMANCE OF THE TRUNCATED NEWTON, CONJUGATE GRADIENT ALGORITHM IN FORTRAN & ADA

by

L.C.W. DIXON & Z.A. MAANY

MARCH 1988

UNITED STATES ARMY

EUROPEAN RESEARCH OFFICE OF THE ARMY

LONDON ENGLAND

CONTRACT NUMBER DAJA45-87-C-0038

THE HATFIELD POLYTECHNIC

PRELIMINARY REPORT NOT FOR DISTRIBUTION

1 INTRODUCTION

This project is concerned with the optimisation of objective functions $F(x)$ in a large dimensional space R^n on highly parallel computers.

It has been established that the truncated Newton method introduced by Dembo & Steihang [1] is an efficient method for solving large optimisation algorithms on a sequential machine, Dixon & Price [2]. The truncated Newton method consists of two main steps

(i) the calculation of the function value $F(x)$, gradient vector $g(x)$ and Hessian matrix $H(x)$ at a sequence of points $x^{(k)}$.

(ii) solving the set of linear equations

$$H(x)\, d = -\, g(x)$$

approximately for the search direction d.

It has been shown Dixon & Mohsenina [3] that the calculation of the gradient vector and Hessian matrix $H(x)$ can be undertaken very elegantly in ADA using automatic differentiation, Rall [4]. In ADA it is only necessary to program the objective function in the normal way, then to declare the variables to be of type "triplet" and to use an extended definition of the arithmetic operators *,+,-,/ to obtain the gradient and Hessian. All the necessary software for triplet and sparse triplet arithmetic has now been written and tested. Sparse triplet arithmetic generates the Hessian in a standard sparse form convenient for large sparse matrices.

In contrast automatic differentiation in Fortran is messy as each arithmetic operation in the calculation of the function must be replaced by a subroutine call to perform the triplet arithmetic. This is not a major problem for the simple test functions used in the tests reported herein,

but would be impractical on realistic industrial problems.

Using automatic differentiation in ADA should remove the difficult task of ensuring that the gradient calculation is coded correctly and also the need to estimate a suitable step with which to approximate the Hessian by differences which is the common practise.

It is anticipated that when the triplet arithmetic is declared to be concurrent tasks then the ADA version should be efficient on highly parallel computers.

In the truncated Newton method the set of equations

$$H(x)\ d = -\ g(x)$$

is usually approached by applying the conjugate direction algorithm. This is an iterative method that decreases the quadratic approximation to F

$$Q(x,U) = F(x) + g^T U + 1/2\ U^T H U$$

and increases $||U||$ at each inner iteration.

The kth inner iteration is terminated when either

$$\text{(i)} \quad ||g(x+U)|| \leq ||g(x)||\ \min\ (0.1/k,\ ||g(x)||)$$

$$\text{or (ii)} \quad ||U|| \geq D.$$

Again the conjugate direction method which consists of updating vectors can readily be posed as in concurrent tasks.

Evidence exists Dixon & Mohsenina [5] that the introduction of an

incomplete Choleski decomposition of the matrix H(x), and performing the conjugate gradient algorithm in the scaled space could be beneficial on simple problems.

The functions reported in [5] were in a sense special as they were both modifications of simple low dimensional problems, that had been extended in such a way that the number of distinct eigenvalues remained low (which favoured the conjugate gradient method) and had a small band width (which favoured a full choleski decomposition).

In wishing to test these ideas further one of the first requirements was to define a more general set of simple test functions that led to sparse Hessians that could be made arbitrarily illconditioned. The set chosen are defined in Appendix 1. The Hessians are very sparse having the familiar diagonal band structure but with some non zero diagonals far from centre. The functions can be made more illconditioned by increasing the power of the $(i/n)^r$ coefficient, to date tests have been performed with $r = 0$, 1 and 2.

2 NUMERICAL RESULTS

The results of the tests using the TNCG code in Fortran are given in Appendix 2. Most of these were terminated when $||g|| \leq 10^{-6}$, though for some illconditioned problems it was necessary to continue until $||g|| \leq 10^{-10}$ to obtain a good approximate result. All the tests were successful the algorithm behaving as expected. The results on the large dimensional illconditioned problems were very expensive in terms of computer time and confirm the need to improve the algorithm in these cases.

The same tests were commenced using ADA, it was found as expected that the same number of outer and inner iterations was usually required, though on occasions one extra or fewer outer iteration was performed. However the test series could not be completed because the ratio of the CPU times in ADA compared with FORTRAN increased rapidly with n. Theoretically the ratio was expected to remain constant. These results which are shown in Appendix 3 seem to indicate that either

(1) the ADA implementation is poor

(2) our ADA compiler is poor

or (3) ADA is not suited to high dimensional problems of this type.

This result was unexpected and is being investigated further.

Initial investigation using the incomplete Choleski code on large problems have indicated that its performance can be poor if the decomposition attempts to introduce a large number of negative diagonal elements. It is intended to introduce Papadrakakis's safeguards [6] to overcome this problem to determine whether the good results be obtained or some special problems will be repeated on this very different test set. However his safeguards are heuristic for general problems and it seemed sensible to investigate a more theoretical approach.

3 THEORETICAL STUDY

Let us assume that the Hessian matrix is A and that R is an approximation to the inverse.

The iterative scheme

$$X^{k+1} = (I - RA)\, x^k + Rb \qquad (2)$$

converges to the solution x^*, if $||I-RA|| < 1$.

This is easily seen if we write it as

$$X^{k+1} - x^* = (x^k - x^*) + R\,(Ax^* - Ax^k)$$

if now e^k is the error

$$e^{k+1} = (I - RA)\, e^k$$

$$||e^{k+1}|| < ||I - RA||\ ||e^k||.$$

The performance will improve in terms of iterations as $||I - RA||$ is decreased; but any matrix R for which $||I - RA|| < 0.5$ will lead to a reasonable speed of convergence. So we might wish to find a sparse matrix R for which $N = ||I - RA|| < 0.5$ for some norm. We note that if $R_{ij} = 0$ then $N = 1$ and if $R = A^{-1}$ then $N = 0$. It should therefore be possible by introducing the variables one at a time to find a sparse matrix R with the required property. The interaction (2) consisting as it does of sparse matrix vector multiplications should be efficient on a highly parallel computer.

There are of course a number of norms that can be considered. Introducing $B = I - RA$, $||B||_2^2 =$ max eigenvalue B^TB, noting that B^TB is symmetric and positive semi definite.

$$T = \text{Trace}\ (B^TB) > ||B||_2^2$$

so we could select R_{ij} to reduce $T =$ trace (B^TB), T is of course a quadratic in R_{ij}

$$B_{ij} = \delta_{ij} - \sum_k R_{ik}\, A_{kj}$$

$$(B^TB)_{lm} = \sum_j B_{jl}\, B_{jm}$$

$$\text{So} \qquad T = \sum_l \left(\sum_j B_{jl}^2\right)$$

$$= \sum_l \sum_j \left(\delta_{jl} - \sum_k R_{jk}\, A_{kl}\right)^2$$

The reduction of T decomposis into n separate problems as $T = \sum_j T_j$ where $T_j = \sum_l \left(\delta_{jl} - \sum_k R_{jk}\, A_{kl}\right)^2$ and R_{jk} only effects T_j.

We can therefore introduce elements R_{jk} in the row R_j to reduce T_j. This is a smooth quadratic function so the introduction of each variable must reduce T_j

$$\text{Viz} \quad R_{jk} = 0 \text{ all } k \qquad T_j = 1$$

$$\text{introduce } R_{jj} = A_{jj}/\sum_k A_{jk}^2 \qquad T_j = 1 - A_{jj}^2/\sum_k A_{jk}^2$$

The method must converge with $T_j = 0$ in n steps at most if at each iteration the parameter R_{jk} are chosen to minimise T_j over the spanned subspace. This method will be investigated further. It is however not clear from the above approach when the process should be terminated. However if instead we look at the $||B||_\infty$ norm

$$||B||_\infty = \max_j \sum_m |B_{jm}| = \max_j U_j$$

$$U_j = \sum_m |B_{jm}| = \sum_m \left|\delta_{jm} - \sum_k R_{jk}\, A_{km}\right|$$

We note that again the elements of each row of R only effect one subfunction so the introduction of elements into the j^{th} row of R may terminate when $U_j < 1/2$.

One approach would therefore consist of introduce the variable R_{jk} one at a time to minimise T_j terminating when $U_j < 1/2$. This naturally raises

the possibility of minimising U_j rather than T_j. U_j is a non differentiable function. Its minimisation can be posed as a linear programming problem. It is however degenerate and it is not necessarily possible in the simplex method to introduce the variables $x_k = R_{jk}$ one at a time and reduce U_j, as the degeneracy of

$$\text{Min} \sum_m u_m + v_m$$

$$\delta_{jm} - \sum_k A_{km} (x_k^+ - x_k^-) + u_m - v_m = 0$$

$$\text{s.t } u_m \geq 0,\ v_m \geq 0,\ x_k^+ \geq 0 \text{ and } x_k^- \geq 0.$$

implies that many zero steps may be taken. These and similar methods involving incomplete Cholesky factors will be investigated further.

4 CONCLUSIONS

(1) A set of test functions have been defined.

(2) Fortran and Ada implementations of the Truncated Newton/Conjugate gradient algorithm have been implemented using automatic differentiations.

(3) The Fortran results were as expected but the ADA results deteriorated as n increased.

(4) Incomplete Choleski versions of both are almost complete.

(5) Methods for finding sparse approximate inverse Hessians have been proposed.

References

1 Dembo R & Steihang T "Truncated Newton Methods for Large Scale Optimisation" Math. Programming Vol 26 1983 pp 190-212.

2 Dixon L C W & Price R "Numerical Experience with the Truncated Newton Method" JOTA Vol 56 No 2 1988.

3 Dixon L C W & Mohsenina M "The Use of the Extended Operator Set of ADA with Automatic Differentiation and the Truncated Newton Method". The Hatfield Polytechnic TR 176 1987.

4 Rall L B "Automatic Differentiation : Techniques & Application" Lecture Notes in Computer Science No 120 Springerverlag 1981

5 Dixon L C W & Mohsenina M "Incomplete Choleski Decomposition in the Truncated Newton Method" 12th IMACS World Congress on Scientific Computers, Paris July 18-22 1988.

6 Papadrakakis M "Accelerating Vector Iteration Methods", J. Appl. Mech. 53 291-292 1986.

APPENDIX 1

The test function used is

$$\text{minimise } f(x) = 1.0 + \sum_{i=1}^{N} 0.5\, a_i x_i^2 + \sum_{i=1}^{N-1} b_i\, (x_i x_{i+1} + x_i\, x_{i+1}^2)^2$$

$$+ \sum_{i=1}^{2n} c_i\, x_i^2\, x_{i+n}^4 + \sum_{i=1}^{n} d_i\, (x_i x_{i+2n})$$

where $N=3n$ = number of optimization variables.

The optimal solution is

$$x_i^* = 0 \qquad i = 1,2 \ldots N$$

and $f(x^*) = 1.0$.

Twelve cases were investigated and they are classified according to the values of a_i, b_i, c_i and d_i as shown in the following table

Case No.	a_i	b_i	c_i	d_i
1	1.0	0.0	0.125	0.125
2	1.0	0.0625	0.0625	0.0625
3	1.0	0.125	0.125	0.125
4	1.0	0.26	0.26	0.26
5	i/N	0.0	0.125	0.125 i/N
6	i/N	0.0625	0.0625	0.0625 i/N
7	i/N	0.125	0.125	0.125 i/N
8	i/N	0.26	0.26	0.26 i/N
9	i^2/N^2	0.0	0.125	0.125 i^2/N^2
10	i^2/N^2	0.0625	0.0625	0.0625 i^2/N^2
11	i^2/N^2	0.125	0.125	0.125 i^2/N^2
12	i^2/N^2	0.26	0.26	0.26 i^2/N^2

APPENDIX 2

This appendix contains the results of the TNCG codes using both Ada and FORTRAN. Table 1 contains the results using Ada for the 12 test cases and N = 15,30,60,90 and 120. For each case the result is given in 2 rows. The first row contains no. of function calls/no. of major iterations/no. of minor interations. The second row contains the CPU time/in seconds. The computer used to run all the Ada test runs is VAX 11/785 using VMS V 4.5.

Table 2 contains the results for the same runs as in table 1 but using FORTRAN. The computer used for the FORTRAN runs is VAX 8650 using VMS V 4.6.

From tables 1 and 2 it can be seen that for most of the cases tested both Ada and FORTRAN codes took the same number of function calls, major iterations and minor iterations. The few cases where there were discrepancies between Ada and FORTRAN are marked with an * in both tables.

The stopping rule used in the above runs was $||\underline{g}|| \leq 10^{-6}$ where $\underline{g}$ is the gradient vector. The CPU time for Ada was large compared with that of FORTRAN. We then decided to run the FORTRAN code for larger values of N. The values N=300, 1500 and 3000 were used. The results for these runs are given in table 3. In this table the output of each run is given in 4 lines. The first line contains no. of function calls/no. of major iteration/no. of minor iterations. The second line contains $\Sigma\ |x_i|$ at the final point. The third line contains the norm of the gradient vector at the final point. The last line contains the CPU time in seconds.

The stopping rule used was $||\underline{g}|| \geq 10^{-6}$. In cases 9-12 for large N $\Sigma|x_i|$ is not near zero as expected. For example $\Sigma|x_i| = 0.9294$ for case 12 with N=1500 and $\Sigma|x_i| = 1.3238$ for case 10 with N = 3000.

To investigate these results we decided to repeat the same test with accuracy of 10^{-10} instead of 10^{-6}. Using 10^{-10} as the required accuracy the above problem disappeared but the CPU time increases. The results for both 10^{-6} and 10^{-10} accuracy are given in table 3.

N	15	30	60	90	120
Case 1	8/6/10 3.07	9/6/10 9.47	9/6/10 31.22	10/7/10 79.47	10/7/10 142.36
Case 2	10/7/11 6.55	11/7/11 19.67	11/7/10 68.27	12/8/11* 174.88	12/8/11* 311.81
Case 3	11/7/12 6.62	11/7/12 19.32	12/8/14 78.46	11/7/11 156.99	11/7/11 279.76
Case 4	15/9/23 8.43	16/10/32 27.16	13/8/13 79.72	13/8/13 178.48	13/8/13 318.09
Case 5	12/8/36 4.08	11/8/59 12.25	12/8/69 41.99	11/8/68 92.68	11/8/79 166.43
Case 6	12/8/37 7.32	12/8/46 22.85	13/9/66 88.68	18/12/136 259.50	13/9/75 349.08
Case 7	14/9/46 8.38	14/9/59 25.21	18/11/83 111.01	15/10/96 218.69	19/12/96 468.09
Case 8	18/10/43 9.67	17/11/52 30.54	18/11/59 109.88	35/17/159 398.44	21/13/141 494.50
Case 9	11/8/50 4.14	11/8/87 12.49	12/9/200 51.84	12/9/270 109.10	12/9/353 195.16
Case 10	15/11/72 9.83	15/10/118 28.82	17/12/289 125.95	28/15/436 359.75	43/21/880* 894.51
Case 11	15/10/70 9.31	21/12/151 36.05	32/17/378* 194.25	30/17/491* 393.29	24/13/417 536.53
Case 12	18/10/59 9.64	22/13/144 37.75	32/15/298 176.52	31/17/487 402.21	40/20/569* 834.23

Table 1 **Results of TNCG using Ada**

*** implies that the results of the Ada code is not the same as that of the FORTRAN code.**

N	15	30	60	90	120
Case 1	8/6/10 0.26	9/6/10 0.55	9/6/10 0.92	10/7/10 1.53	10/7/10 2.02
Case 2	10/7/11 0.44	11/7/11 0.92	11/7/10 1.77	11/7/8* 2.71	11/7/8* 3.61
Case 3	11/7/12 0.44	11/7/12 0.92	12/8/14 2.00	11/7/11 2.78	11/7/11 3.66
Case 4	15/9/23 0.64	16/10/32 1.42	13/8/13 2.09	13/8/13 3.04	13/8/13 4.02
Case 5	12/8/36 0.38	11/8/59 0.90	12/8/69 1.68	11/8/68 2.57	11/8/79 3.51
Case 6	12/8/37 0.60	12/8/46 1.14	13/9/66 2.85	18/12/136 6.44	13/9/75 5.99
Case 7	14/9/46 0.72	14/9/59 1.48	18/11/83 3.65	15/10/96 5.06	19/12/96 7.85
Case 8	18/10/43 0.76	17/11/52 1.58	18/11/59 3.30	35/17/159 9.14	21/13/141 9.57
Case 9	11/8/50 0.40	11/8/87 0.96	12/9/200 2.88	12/9/268 5.38	12/9/351 8.64
Case 10	15/11/72 0.86	15/10/118 1.93	17/12/287 5.98	28/15/433 12.83	40/20/762* 27.03
Case 11	15/10/70 0.84	21/12/151 2.34	26/15/371* 7.98	29/16/383* 12.26	24/13/415 15.52
Case 12	18/10/59 0.82	22/13/144 2.28	32/15/286 7.27	31/17/473 14.19	50/24/904* 32.05

Table 2 **Results of TNCG using FORTRAN**

*** implies that the results of the Ada code is not the same as that of the FORTRAN code.**

N	300		1500		3000	
Accuracy	10^{-6}	10^{-10}	10^{-6}	10^{-10}	10^{-6}	10^{-10}
Case 1	10/6/9 $4.4\ 10^{-18}$ $2.7\ 10^{-20}$ 4.52	same 4.56	11/7/10 $5.3\ 10^{-18}$ $6.8\ 10^{-21}$ 25.46	same 25.41	13/8/11 $4.2\ 10^{-18}$ $2.5\ 10^{-21}$ 57.32	same 57.36
Case 2	11/7/8 $1.9\ 10^{-5}$ $2.0\ 10^{-7}$ 8.81	12/8/11 $2.5\ 10^{-19}$ $2.9\ 10^{-21}$ 9.90	14/9/12 $1.5\ 10^{-18}$ $2.1\ 10^{-19}$ 56.57	same 58.70	14/9/12 $2.6\ 10^{-17}$ $4.7\ 10^{-18}$ 106.46	same 113.37
Case 3	12/8/12 $2.5\ 10^{-13}$ $5.9\ 10^{-14}$ 10.19	same 9.88	13/8/10 $4.0\ 10^{-5}$ $1.8\ 10^{-7}$ 50.75	14/9/13 $2.0\ 10^{-19}$ $1.6\ 10^{-21}$ 55.97	14/9/13 $5.1\ 10^{-13}$ $1.2\ 10^{-13}$ 109.71	same 113.07
Case 4	13/8/12 $5.6\ 10^{-8}$ $1.0\ 10^{-8}$ 10.35	14/9/15 $1.3\ 10^{-23}$ $2.3\ 10^{-24}$ 11.26	15/9/13 $1.5\ 10^{-10}$ $2.9\ 10^{-11}$ 57.50	same 57.14	15/9/13 $1.4\ 10^{-7}$ $1.9\ 10^{-8}$ 113.29	16/10/16 $1.5\ 10^{-22}$ $1.9\ 10^{-23}$ 125.92
Case 5	12/8/136 $5.1\ 10^{-7}$ $1.2\ 10^{-9}$ 11.65	14/9/231 $1.7\ 10^{-15}$ $9.4\ 10^{-18}$ 16.11	13/9/274 $3.6\ 10^{-6}$ $1.3\ 10^{-9}$ 91.68	15/10/484 $2.8\ 10^{-14}$ $1.8\ 10^{-17}$ 140.88	14/9/240 $2.0\ 10^{-13}$ $3.9\ 10^{-7}$ 164.55	15/10/473 $3.9\ 10^{-9}$ $1.3\ 10^{-12}$ 275.97
Case 6	19/12/214 $2.6\ 10^{-8}$ $6.1\ 10^{-11}$ 26.08	same 25.57	37/19/604 $7.0\ 10^{-6}$ $1.6\ 10^{-8}$ 288.49	38/20/779 $6.7\ 10^{-14}$ $4.6\ 10^{-17}$ 333.86	50/21/814 $6.5\ 10^{-7}$ $1.5\ 10^{-9}$ 699.62	60/22/1450 $6.5\ 10^{-7}$ $1.5\ 10^{-9}$ 1084.94
Case 7	20/13/177 $1.2\ 10^{-4}$ $9.2\ 10^{-7}$ 24.59	21/14/247 $6.9\ 10^{-11}$ $2.7\ 10^{-13}$ 29.63	37/16/453 $6.1\ 10^{-4}$ $7.0\ 10^{-7}$ 232.45	38/17.592 $5.9\ 10^{-9}$ $8.7\ 10^{-12}$ 272.12	42/19/764 $1.7\ 10^{-4}$ $2.3\ 10^{-7}$ 647.87	43/20/983 $1.0\ 10^{-10}$ $1.0\ 10^{-13}$ 800.54

Case 8	37/16/233 $3.5\ 10^{-6}$ $9.8\ 10^{-9}$ 33.98	38/17/327 $3.3\ 10^{-14}$ $1.5\ 10^{-16}$ 40.86	33/17/523 $7.5\ 10^{-6}$ $1.8\ 10^{-10}$ 253.96	43/18/1051 $7.5\ 10^{-6}$ $1.8\ 10^{-10}$ 385.80	38/19/713 $1.0\ 10^{-6}$ $9.5\ 10^{-10}$ 608.76	48/20/1353 $1.0\ 10^{-6}$ $9.5\ 10^{-10}$ 983.55
Case 9	14/9/835 $2.1\ 10^{-6}$ $6.4\ 10^{-9}$ 42.80	15/10/1265 $6.9\ 10^{-16}$ $1.3\ 10^{-17}$ 62.51	14/10/3340 $1.6\ 10^{-3}$ $4.6\ 10^{-7}$ 793.80	15/11/5614 $8.5\ 10^{-12}$ $3.0\ 10^{-14}$ 1291.51	14/9/4799 $5.3\ 10^{-3}$ $2.3\ 10^{-7}$ 2151.60	15/10/9409 $8.9\ 10^{-11}$ $2.8\ 10^{-13}$ 4276.02
Case 10	37/20/1995 $3.9\ 10^{-3}$ $1.6\ 10^{-7}$ 132.56	38/21/2395 $1.0\ 10^{-6}$ $5.6\ 10^{-11}$ 154.44	58/31/7008 0.066 $3.2\ 10^{-7}$ 2153.41	61/34/15738 $5.5\ 10^{-9}$ $2.8\ 10^{-14}$ 4393.73	65/32/3216 1.3228 $9.5\ 10^{-7}$ 2095.39	70/37/21944 $1.9\ 10^{-8}$ $2.4\ 10^{-14}$ 11942.35
Case 11	51/24/1666 $6.6\ 10^{-3}$ $4.2\ 10^{-7}$ 121.44	53/26/2393 $2.0\ 10^{-9}$ $1.4\ 10^{-13}$ 161.36	80/35/3692 0.4611 $9.4\ 10^{-7}$ 1263.18	86/40/12870 $4.7\ 10^{-9}$ $3.9\ 10^{-14}$ 3638.44	69/32/2251 1.4219 $6.9\ 10^{-7}$ 1570.84	77/38/29467 $5.1\ 10^{-5}$ $2.2\ 10^{-11}$ 15189.91
Case 12	53/26/1596 $2.3\ 10^{-4}$ $8.8\ 10^{-4}$ 120.09	54/27/2015 $4.8\ 10^{-10}$ $2.8\ 10^{-13}$ 141.73	80/37/2435 0.9294 $1.0\ 10^{-6}$ 908.40	97/48/13019 $3.4\ 10^{-8}$ $3.0\ 10^{-13}$ 3790.82	81/22/3880 1.1997 $7.3\ 10^{-7}$ 2516.05	39/64/24144 $2.0\ 10^{-6}$ $1.9\ 10^{-11}$ 13462.14

Table 3 Results of TNCG for large N using FORTRAN

APPENDIX 3

The tables of results in Appendix 2 indicate that among the first four Cases, Case 4 is harder than Case 3 which in turn is harder than Case 2, while Case 2 is harder than Case 1. The same seems also true for Cases 8,7,6 and 5 and for the last set of Case 12,11,10 and 9. These results were expected when we constructed the test problems. We decided to study the performance of Case 1,4,5,8,9 and 12 in more detail.

In figure 1 CPU time per major iterations, is plotted against N when using the FORTRAN code, for N in the range (15-120).

Figure 2 shows the plot for N in the range (15-3000). Both these figures shows that CPU time/major iterations is linear in N for Case 1,4,5 and 8, while for Cases 9 and 12 the function is non-linear. The same plot was repeated for the Ada Code and the result is given in figure 3. In this figure none of the cases is a linear relationship. It must be stressed that the figures 1 and 3 the scale of CPU time/major iterations is not the same since the FORTRAN Code is much faster than the Ada Code. Figure 4 shows CPU time/major iterations plotted against N for both the FORTRAN and Ada runs. The FORTRAN and the CPU times were multiplied by 10 in this figure.

For the FORTRAN Code it seems that the relation is not linear because we have ignored the effect of the changing number of minor iterations. For cases 1-4 the ratio minor iteration/major iteration is less than 3, for cases 5-8 this ratio becomes as large as 30, while for cases 9-12 this ratio reaches more than 500. In cases 9-12 the minor iterations play an important role which can not be ignored.

As for this problem, on average, one major iteration costs the same as about 30 minor iterations. Using equivalent iteration = major iteration + minor iterations/30 we draw CPU time/equivalent iterations against N. Figures 5 and 6 show these plots for the FORTRAN Codes. These figures show that the relation is now virtually linear. The same plot for Ada is given in figure 7 which indicates that none of the cases are linear. Again the scale of CPU time/equivalent iteration used in figures 5 and 7 is not the same since the FORTRAN Code was much faster than the Ada one. To show the relative time between FORTRAN and Ada, figure 8 contains the plots for both

FORTRAN and Ada. In this figure the FORTRAN CPU time is multiplied by 10 to separate the different cases.

These figures show that the performance of the Ada Code needs more investigation as it is obviously dominated by the non-linear factor that is absent in the FORTRAN implementation.

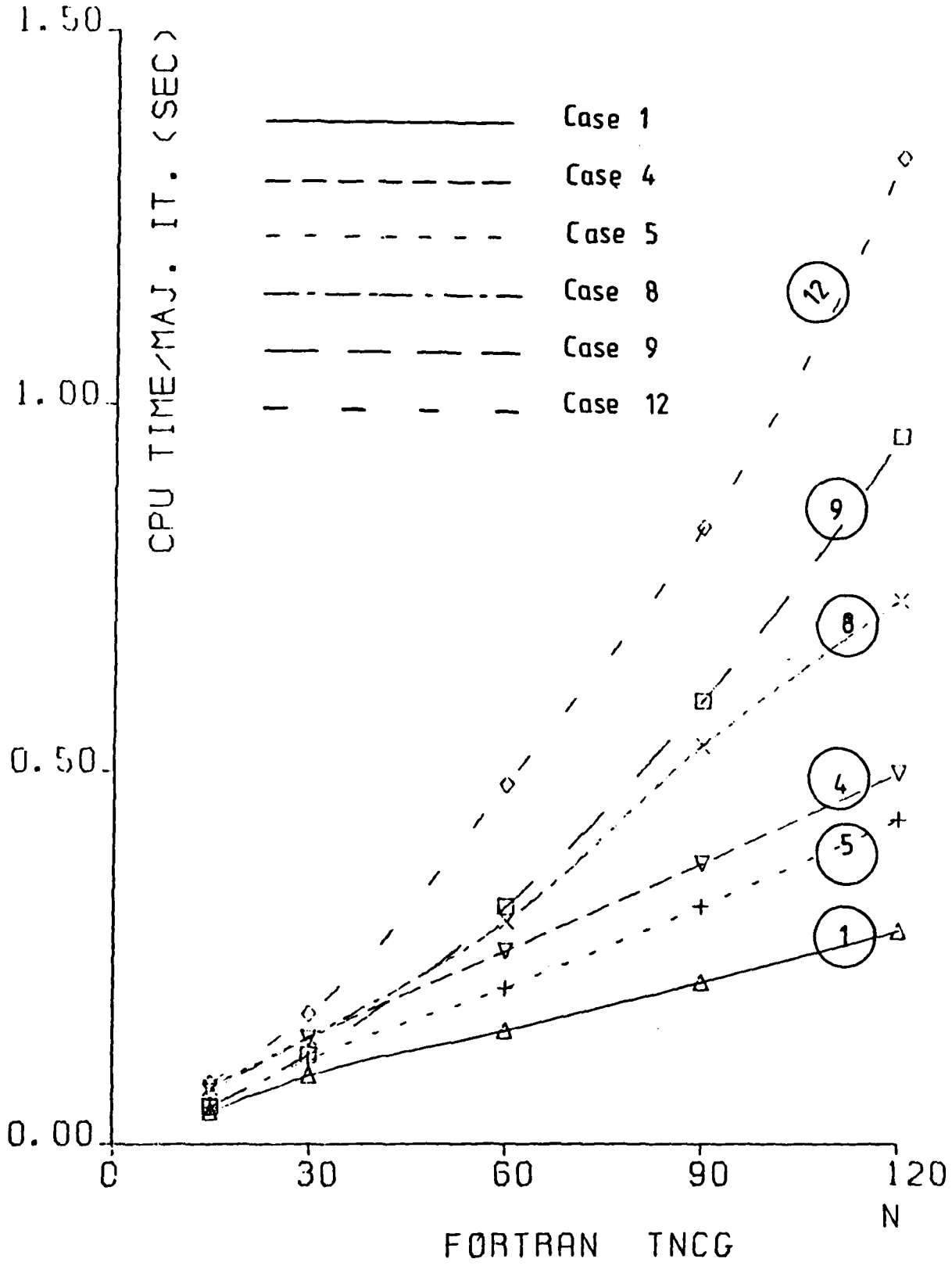


Figure 1 CPU time/major iteration "FORTRAN Code"

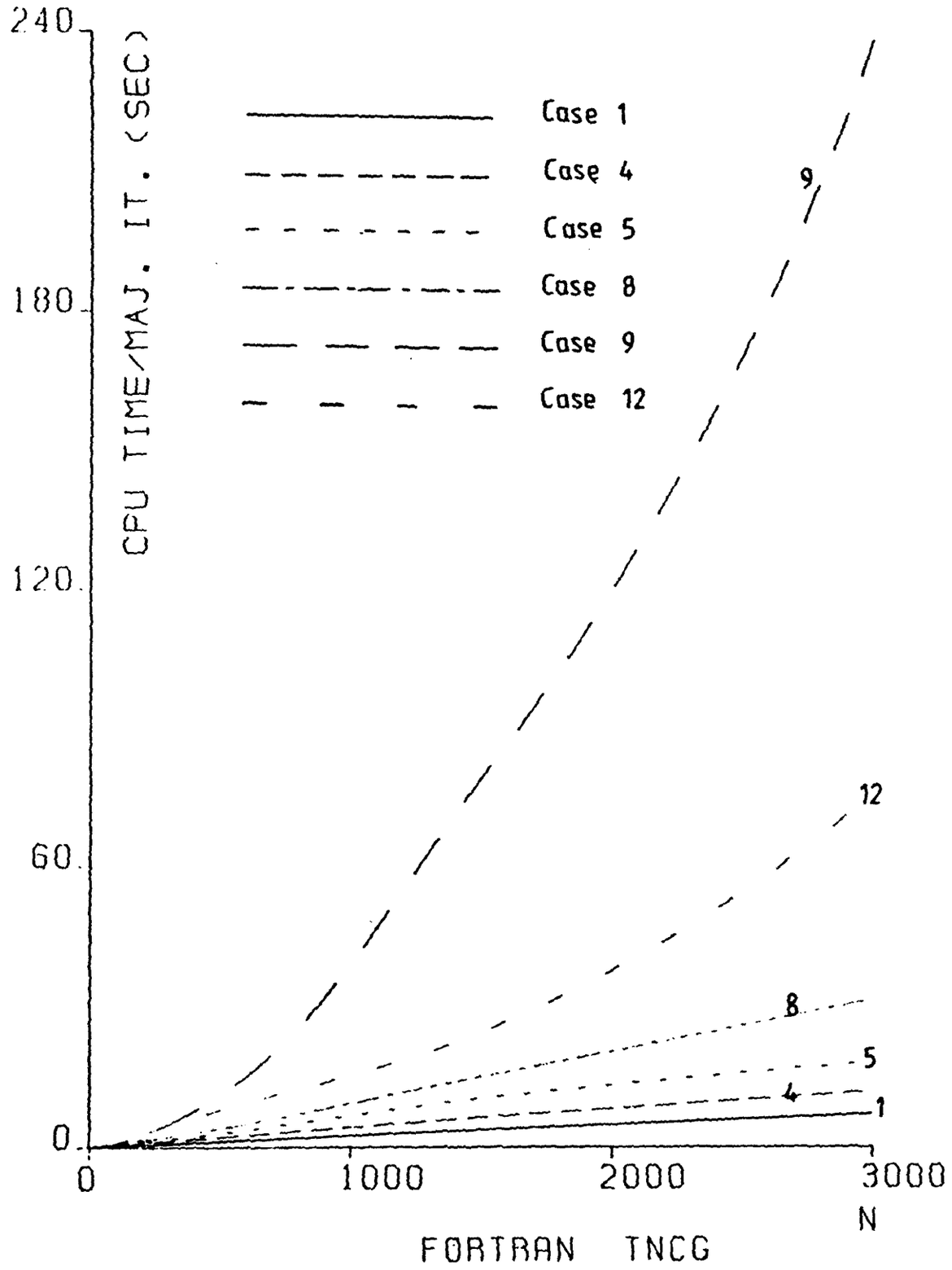


Figure 2 CPU time/major iteration "FORTRAN Code/large N"

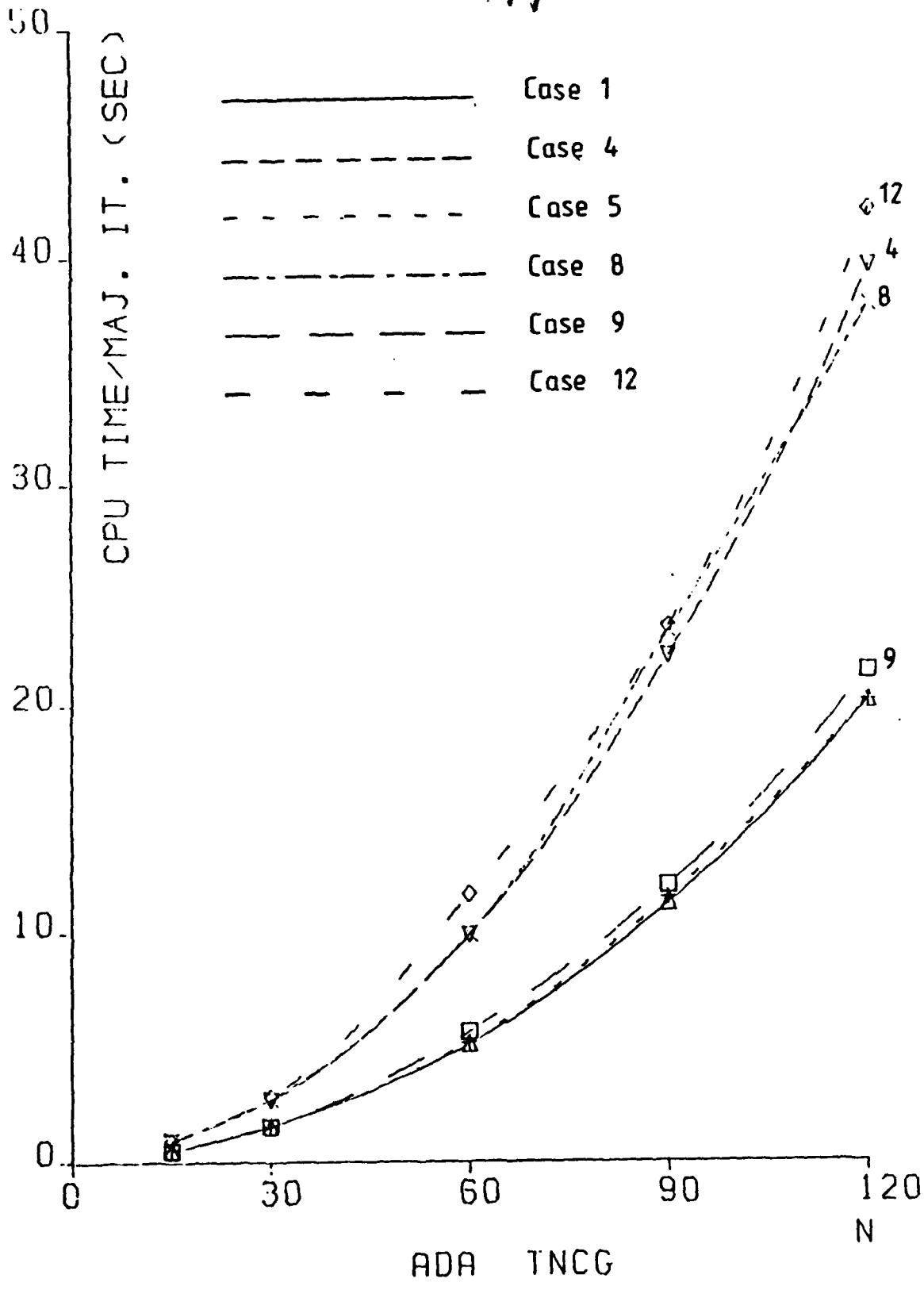


Figure 3 CPU time/major iteration "Ada Code"

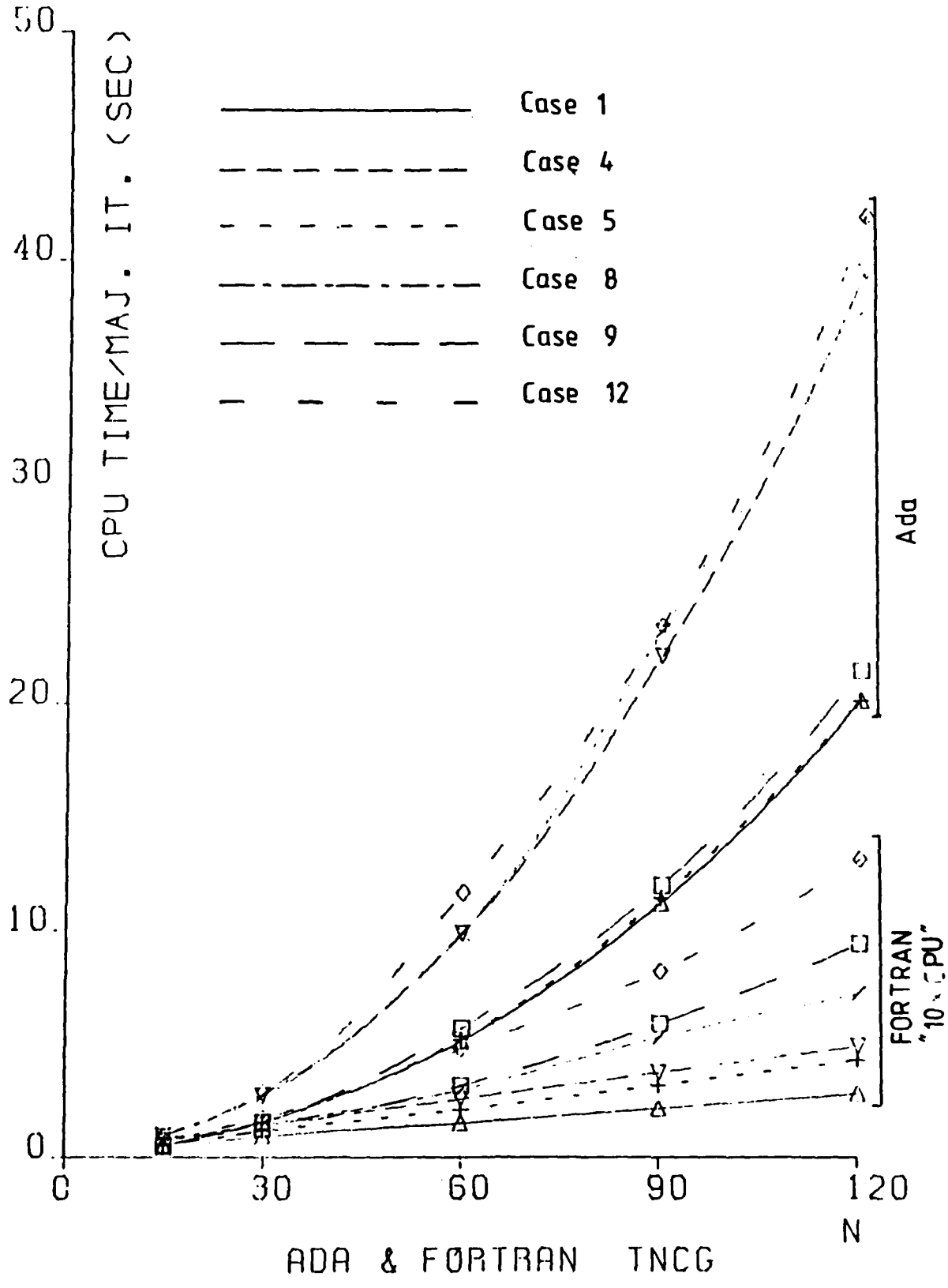


Figure 4 CPU time/major iteration "FORTRAN and Ada Codes"

p.s. for FORTRAN Code the CPU was multiplied by 10.

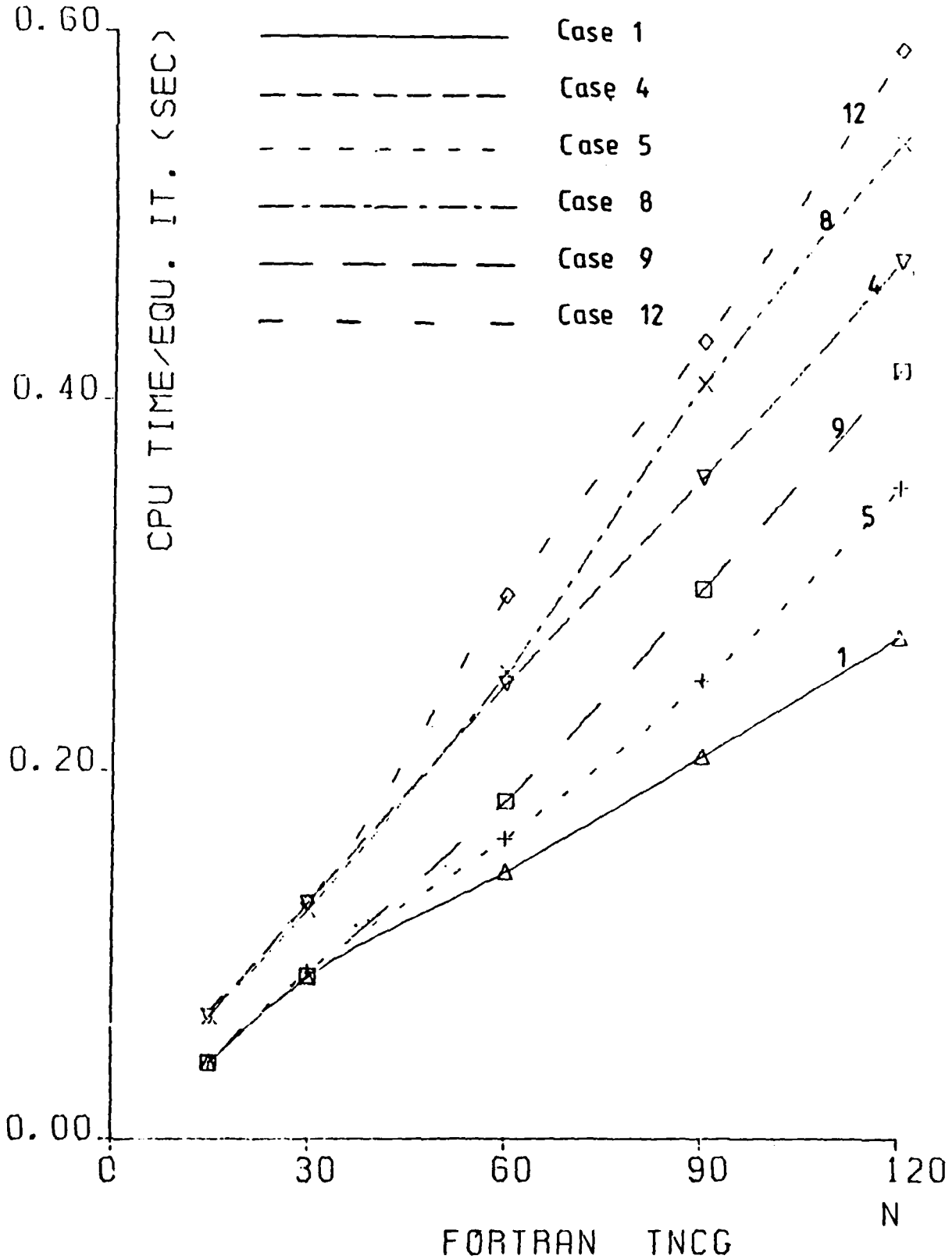


Figure 5 CPU time/equivalent iterations "FORTRAN Code"

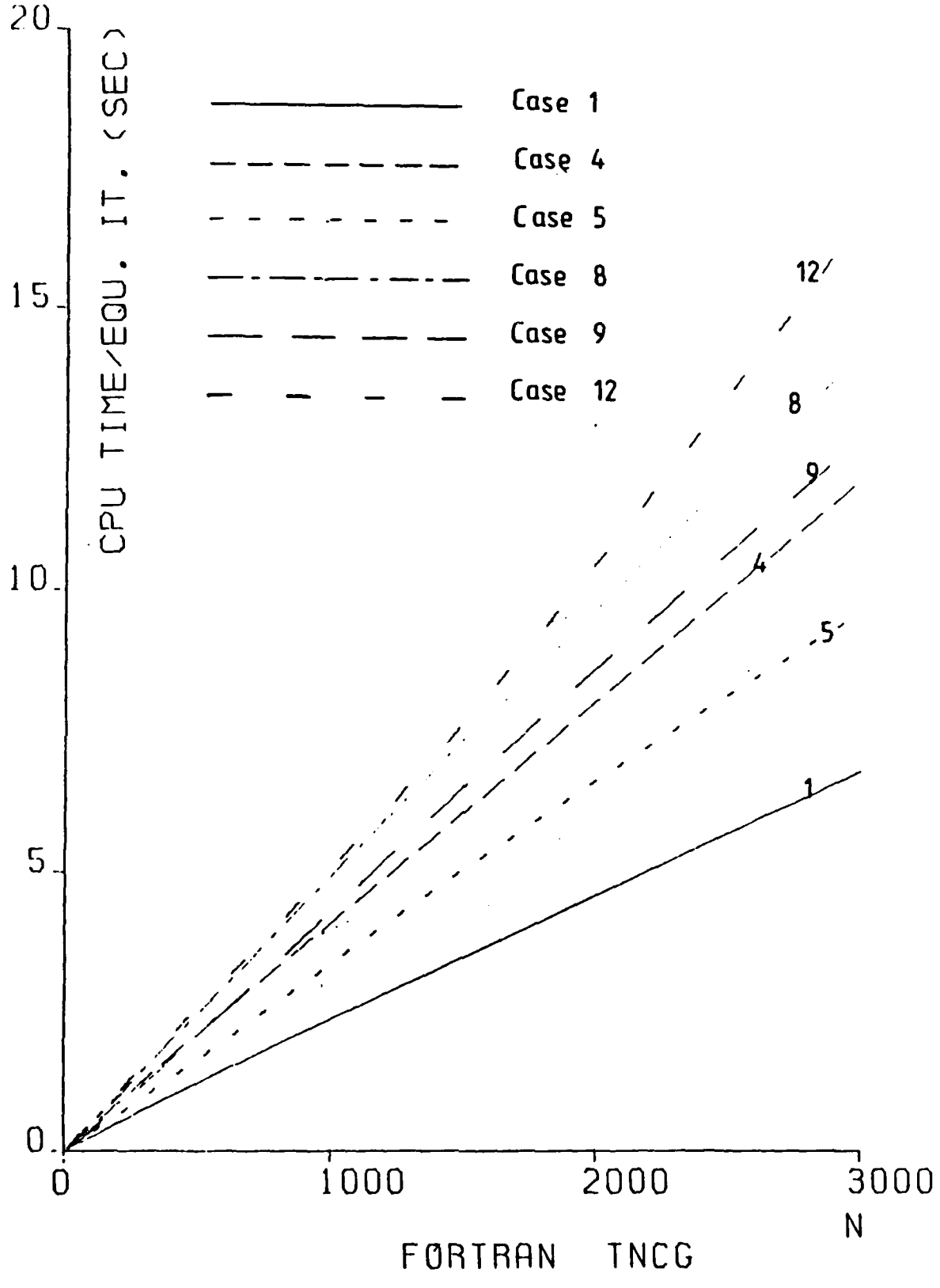


Figure 6 CPU time/equivalent iteration "FORTRAN Code/large N"

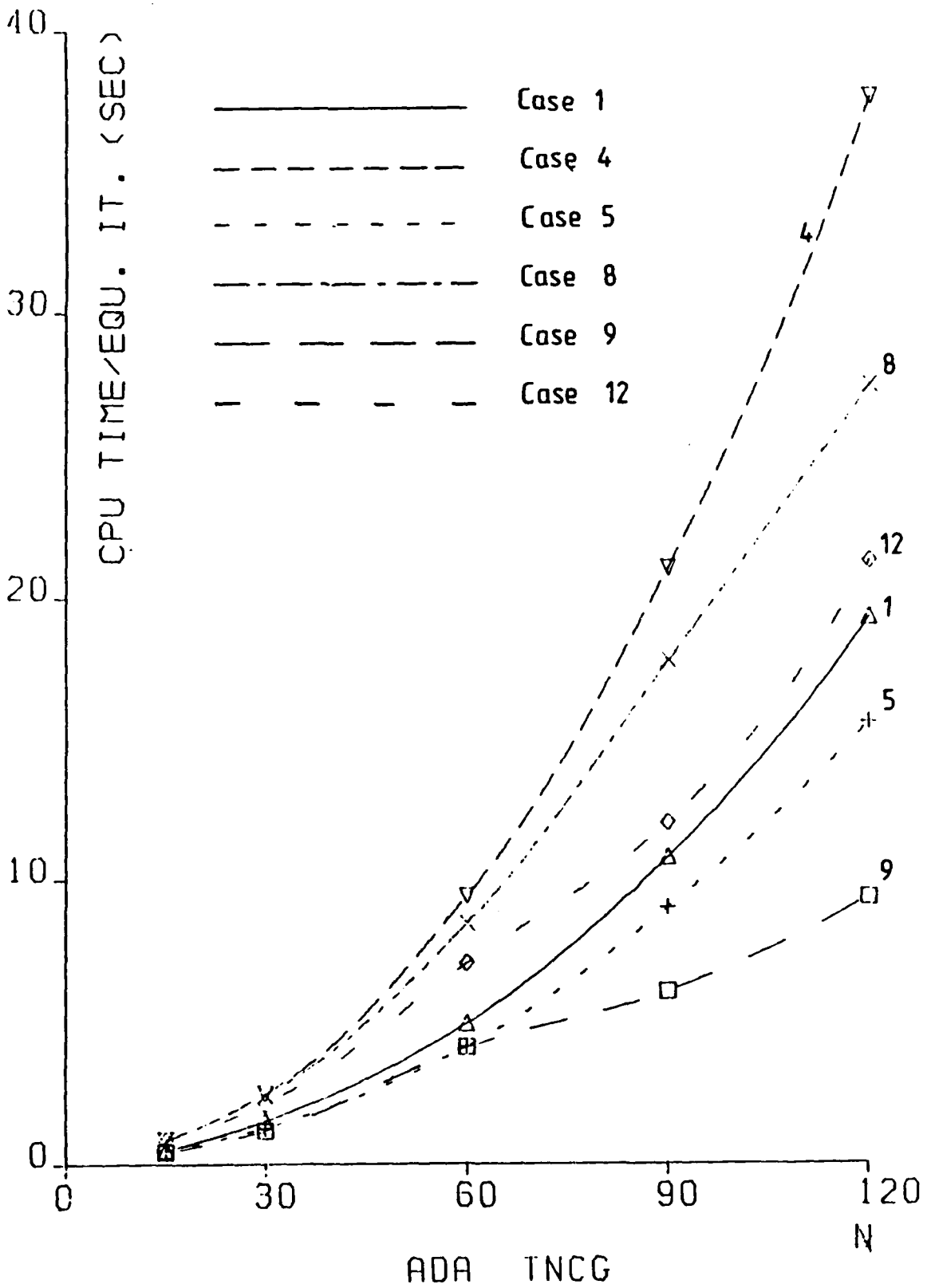


Figure 7 CPU time/equivalent iterations "Ada Code"

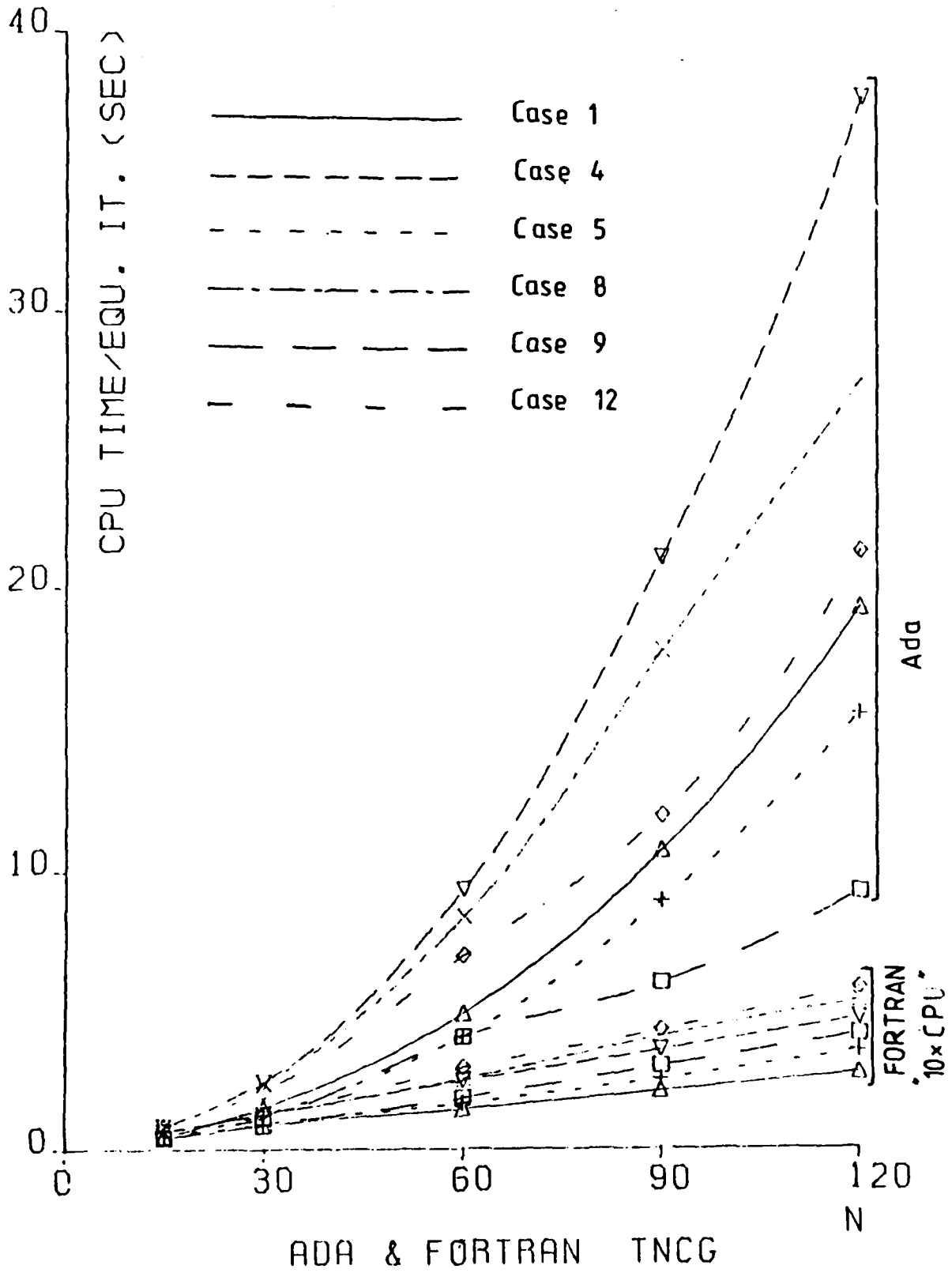


Figure 8 CPU time/equivalent iteration "FORTRAN and Ada Codes"
p.s. for FORTRAN Code the CPU was multiplied by 10.

END
DATE
FILMED
7-88
DTIC